Comprehensive Guide to Siamese CNN for Signature Verification

Introduction

This project implements a **Siamese Neural Network** using **Convolutional Neural Networks (CNN)** for handwritten signature verification. The goal is to train a model that can determine whether two signatures belong to the same person or different people.

1. Installing Required Libraries

```python
!pip install --quiet tensorflow matplotlib pillow opencv-python kagglehub
```

Explanation: This command installs the essential libraries needed for the project:

- **TensorFlow**: The main deep learning framework for building and training the model
- **Matplotlib**: For plotting graphs and visualizing results
- **Pillow**: For image processing and manipulation
- **OpenCV**: For advanced image processing operations
- **KaggleHub**: For downloading datasets from Kaggle

Why do we need these libraries? Each library serves a specific purpose in the project. TensorFlow is the core of the project, while Pillow and OpenCV help process signature images before feeding them to the model.

2. Importing Libraries

```python
import os
import random
import numpy as np
from glob import glob
from PIL import Image
import tensorflow as tf
from tensorflow.keras import layers, models
import kagglehub
```

Explanation:

- **os**: For handling file paths and operating system operations
- **random**: For selecting random samples from the data

- **numpy**: For working with numerical arrays
- **glob**: For searching files using specific patterns
- **PIL.Image**: For reading and converting images
- **tensorflow.keras**: For building the model and its layers

3. Downloading Data from Kaggle

```python
dataset_dir = kagglehub.dataset_download("divyanshrai/handwritten-signatures")
```

Explanation: This line downloads a dataset containing handwritten signature images from Kaggle. The dataset contains signatures from different writers, where each signature has a filename that starts with a number or code identifying the writer.

Why use Kaggle? Kaggle provides ready-made and organized datasets for machine learning, saving time in data collection and organization.

4. Collecting All Image Paths

```python
def collect_images(base_dir):
    exts = ['*.png', '*.jpg', '*.jpeg']
    images = []
    for ext in exts:
        images.extend(glob(os.path.join(base_dir, '**', ext), recursive=True))
    return images

all_imgs = collect_images(dataset_dir)
```

Explanation: This function searches through all folders and subfolders for image files with different formats (PNG, JPG, JPEG). The `**` symbol means searching through all directory levels recursively.

Why search for multiple formats? Signature images might be saved in different formats, and we want to collect all available images regardless of their file extension.

5. Organizing Images by Writer

```python
```

```
writers_dict = {}
for img_path in all_imgs:
    filename = os.path.basename(img_path)
    writer_id = filename.split("_")[0]
    writers_dict.setdefault(writer_id, []).append(img_path)

writers = list(writers_dict.keys())
```

Explanation: This code organizes images into a dictionary where:

- **Key**: The writer's ID number or code
- **Value**: A list of all image paths for that writer's signatures

The writer's identity is extracted from the filename by splitting the text at the "_" character and taking the first part.

Importance of this organization: Essential for creating image pairs later - we need to know which images belong to the same writer to create positive pairs (same writer) and negative pairs (different writers).

6. Image Preprocessing

```python
IMG_SIZE = (100, 100)
def load_img(path):
    img = Image.open(path).convert('L').resize(IMG_SIZE)
    arr = np.array(img) / 255.0
    return np.expand_dims(arr, axis=-1)
```

Explanation: This function processes each image through the following steps:

6.1 Setting Image Size

`IMG_SIZE = (100, 100)` defines that all images will be resized to 100×100 pixels.

6.2 Processing Steps:

1. **Open Image**: `Image.open(path)` reads the image from the file path
2. **Convert to Grayscale**: `convert('L')` converts the image to grayscale
3. **Resize**: `resize(IMG_SIZE)` makes all images the same size
4. **Convert to Array**: `np.array(img)` converts the image to a numerical array
5. **Normalization**: `/ 255.0` scales pixel values from 0-255 range to 0-1 range
6. **Add Dimension**: `expand_dims` adds an extra dimension for the channel

Why these steps?

- **Grayscale**: Signatures are typically black and white, so we don't need color information
- **Uniform Size**: The model requires all images to have the same dimensions
- **Normalization**: Helps the model learn better and faster by keeping values in a manageable range
- **Extra Dimension**: TensorFlow expects images in the format (height, width, channels)

7. Image Pair Generator

```python
def pair_generator(batch_size=32):
    while True:
        X1, X2, y = [], [], []
        for _ in range(batch_size):
            if random.random() < 0.5:
                # Positive pair (same writer)
                w = random.choice(writers)
                if len(writers_dict[w]) < 2:
                    continue
                imgs = random.sample(writers_dict[w], 2)
                label = 1
            else:
                # Negative pair (different writers)
                if len(writers) < 2:
                    continue
                w1, w2 = random.sample(writers, 2)
                imgs = [random.choice(writers_dict[w1]), random.choice(writers_dict[w2])]
                label = 0

            X1.append(load_img(imgs[0]))
            X2.append(load_img(imgs[1]))
            y.append(label)

        yield (np.array(X1, dtype=np.float32), np.array(X2, dtype=np.float32)), np.array(y, dtype=np.float32)
```

Explanation:

7.1 Generator Concept

A generator is a function that produces data incrementally instead of loading everything into memory at once. This is useful when dealing with large datasets that might not fit in memory.

7.2 Creating Image Pairs

The function creates two types of image pairs:

Positive Pairs (label = 1):

- Selects a random writer
- Chooses two different images from the same writer's signatures
- Assigns label = 1 meaning "same writer"

Negative Pairs (label = 0):

- Selects two different writers randomly
- Chooses one image from each writer
- Assigns label = 0 meaning "different writers"

7.3 Data Balance

`random.random() < 0.5` ensures that 50% of pairs are positive and 50% are negative, providing balance in training data.

Why is this balance important? If most data is of one type, the model will learn to be biased toward that type and won't give accurate results on balanced test data.

7.4 Error Handling

The `continue` statements skip cases where a writer has fewer than 2 signatures or there are fewer than 2 writers total, preventing errors.

8. Creating TensorFlow Dataset

```python
BATCH_SIZE = 16
train_dataset = tf.data.Dataset.from_generator(
    lambda: pair_generator(BATCH_SIZE),
    output_signature=(
        (tf.TensorSpec(shape=(None, 100, 100, 1), dtype=tf.float32),
         tf.TensorSpec(shape=(None, 100, 100, 1), dtype=tf.float32)),
        tf.TensorSpec(shape=(None,), dtype=tf.float32)
    )
)
```

Explanation:

8.1 Batch Size Definition

`BATCH_SIZE = 16` means the model will process 16 image pairs at a time during training.

8.2 TensorSpec

`TensorSpec` defines the shape and data type expected:

- `(None, 100, 100, 1)`: None means any number of images, 100×100 is image size, 1 for grayscale
- `dtype=tf.float32`: Data type specification

Why use tf.data.Dataset? It provides an efficient way to feed data to the model with automatic optimization and acceleration capabilities. It also handles batching and can prefetch data for better performance.

9. Building the Base CNN Network

```python
def build_base(input_shape=(100,100,1)):
    model = models.Sequential([
        layers.Input(shape=input_shape),
        layers.Conv2D(32, (3,3), activation="relu"),
        layers.MaxPooling2D((2,2)),
        layers.Conv2D(64, (3,3), activation="relu"),
        layers.MaxPooling2D((2,2)),
        layers.Flatten(),
        layers.Dense(128, activation="relu")
    ])
    return model
```

Explanation:

9.1 Network Architecture

This function builds the base network that will extract features from each signature image.

9.2 Layer-by-Layer Breakdown:

Input Layer:

- Defines the input data shape: (100, 100, 1)

First Convolutional Layer:

- `Conv2D(32, (3,3), activation="relu")`
- Uses 32 filters of size 3×3
- Detects simple features like lines and edges
- **ReLU activation** removes negative values and introduces non-linearity

First Pooling Layer:

- `MaxPooling2D((2,2))`
- Reduces image size by half
- Keeps only the most important information
- Reduces memory and computational requirements

Second Convolutional Layer:

- `Conv2D(64, (3,3), activation="relu")`
- Uses 64 filters (more than the first layer)
- Detects more complex features and patterns

Second Pooling Layer:

- Reduces size again, creating more abstract representations

Flatten Layer:

- Converts the 2D feature maps into a 1D vector
- Necessary before connecting to fully connected layers

Dense Layer:

- `Dense(128, activation="relu")`
- Produces 128 features that represent the signature
- These features are the "digital fingerprint" of the signature

Why this design? The architecture follows the typical CNN pattern: starting with few filters for simple features, then increasing the number for complex features. MaxPooling gradually reduces size while preserving important information.

10. Constructing the Siamese Network

```python
base_net = build_base()
input_a = layers.Input(shape=(100,100,1))
input_b = layers.Input(shape=(100,100,1))
feat_a = base_net(input_a)
feat_b = base_net(input_b)
distance = layers.Lambda(lambda tensors: tf.abs(tensors[0] - tensors[1]))([feat_a, feat_b])
output = layers.Dense(1, activation="sigmoid")(distance)
model = models.Model([input_a, input_b], output)
```

Explanation:

10.1 Siamese Network Concept

A Siamese network consists of two identical networks (sharing the same weights) that process two separate inputs, then compare their outputs.

10.2 Network Components:

Dual Inputs:

- `input_a` and `input_b`: Two separate inputs for the two signature images

Shared Network:

- `base_net` is applied to both images
- **Weight Sharing**: This is the core of Siamese networks - the same network processes both images

Feature Extraction:

- `feat_a = base_net(input_a)`: Extracts features from first image
- `feat_b = base_net(input_b)`: Extracts features from second image

Distance Calculation:

- `tf.abs(tensors[0] - tensors[1])` computes the absolute difference between features
- Smaller distance means more similar images
- This creates a "similarity vector" between the two signatures

Output Layer:

- `Dense(1, activation="sigmoid")` produces a value between 0 and 1
- Close to 1 means "same writer", close to 0 means "different writers"

Why Siamese Network? It learns to measure similarity between images rather than classifying each image separately. This is perfect when we have new writers that the model hasn't seen during training.

10.3 Lambda Layer

The `Lambda` layer allows custom mathematical operations. Here it computes the element-wise absolute difference between the two feature vectors.

11. Model Compilation

python

```python
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Explanation:

11.1 Loss Function

Binary Crossentropy:

- Suitable for binary classification problems (0 or 1)
- Measures the difference between actual and predicted probabilities
- Gives larger penalties for confident wrong predictions
- Formula encourages the model to be confident about correct predictions

11.2 Optimizer

Adam:

- An advanced optimizer that combines advantages of several other optimizers
- Automatically adapts the learning rate during training
- Works well with deep networks and sparse gradients
- Maintains separate learning rates for each parameter

11.3 Metrics

Accuracy:

- Measures the percentage of correct predictions
- Easy to interpret and understand
- Good for balanced datasets (which we have due to 50-50 positive-negative pairs)

12. Model Training

```python
STEPS = 50
VAL_STEPS = 10
EPOCHS = 5

history = model.fit(
    train_dataset,
    steps_per_epoch=STEPS,
    validation_data=train_dataset,
    validation_steps=VAL_STEPS,
    epochs=EPOCHS
)
```

Explanation:

12.1 Training Parameters

Steps per Epoch:

- `STEPS = 50` means 50 batches of data per epoch
- With `BATCH_SIZE = 16`, this means 50 × 16 = 800 image pairs per epoch

Validation Steps:

- `VAL_STEPS = 10` means 10 batches for validation
- This gives 10 × 16 = 160 pairs for validation per epoch

Epochs:

- `EPOCHS = 5` means the training process will repeat 5 times
- Total training samples: 5 × 800 = 4,000 image pairs

12.2 Training Process

Training Data:

- Uses the generator to create fresh pairs each epoch
- Ensures the model sees different combinations of signatures

Validation Data:

- Uses the same generator (in practice, should use separate validation data)
- Monitors performance on "unseen" data to detect overfitting

12.3 Why These Numbers?

- **Small steps/epochs**: Good for initial testing and quick results
- **Can be increased**: For better performance, increase epochs and steps
- **Computational efficiency**: Balanced between training time and performance

13. Results and Accuracy

```python
final_train_acc = history.history['accuracy'][-1]
final_val_acc = history.history['val_accuracy'][-1]
print(f"Final Training Accuracy: {final_train_acc*100:.2f}%")
print(f"Final Validation Accuracy: {final_val_acc*100:.2f}%")
```

Explanation:

13.1 Extracting Results

- `history.history['accuracy'][-1]`: Gets the last training accuracy value

- `history.history['val_accuracy'][-1]`: Gets the last validation accuracy value
- `[-1]` index gets the final epoch's results

13.2 Interpreting Accuracy

- **Training Accuracy**: How well the model performs on data it has seen
- **Validation Accuracy**: How well it performs on validation data
- **Good Signs**: Both accuracies should be reasonably high (>70-80%)
- **Overfitting Warning**: If training accuracy is much higher than validation accuracy

13.3 What the Percentages Mean

- **Above 80%**: Good performance for signature verification
- **50-60%**: Poor performance, model is barely better than random guessing
- **Below 50%**: Very poor performance, model might be learning the wrong patterns

Background Concepts

Siamese Networks

What are they? Siamese networks are neural networks that learn to compare two inputs and determine their similarity. They're called "Siamese" because they use two identical "twin" networks that share the same weights.

Key Advantages:

1. **Few-shot Learning**: Can work with limited data per class
2. **Generalization**: Can handle new writers not seen during training
3. **Similarity Learning**: Learns meaningful representations for comparison

How they work:

1. Two inputs go through identical networks
2. Features are extracted from both inputs
3. A distance/similarity measure is computed
4. A final decision layer determines if inputs are similar or different

Convolutional Neural Networks (CNNs)

Why use CNNs for images?

- **Local Feature Detection**: Conv2D layers detect local patterns like strokes and curves
- **Translation Invariance**: Can recognize features regardless of their position in the image
- **Hierarchical Learning**: Early layers detect simple features, deeper layers detect complex patterns

- **Parameter Sharing**: Same filters are applied across the entire image, reducing overfitting

Image Preprocessing Importance

Normalization (dividing by 255):

- Converts pixel values from 0-255 range to 0-1 range
- Helps gradient descent converge faster
- Prevents certain features from dominating due to scale

Grayscale Conversion:

- Reduces data dimensionality
- Focuses on structure rather than color
- Signatures are typically monochrome anyway

Resizing:

- Ensures consistent input size for the neural network
- Balances between preserving detail and computational efficiency

Loss Function: Binary Crossentropy

Mathematical Background: Binary crossentropy measures how far the predicted probability is from the actual label. For a single sample:

- Loss = -[y × log(p) + (1-y) × log(1-p)]
- Where y is the true label (0 or 1) and p is the predicted probability

Why it works well:

- Heavily penalizes confident wrong predictions
- Encourages the model to be confident about correct predictions
- Smooth gradients help with training stability

Distance Calculation

Absolute Difference: The `tf.abs(tensors[0] - tensors[1])` operation computes element-wise absolute difference between feature vectors. This creates a new vector where each element represents how different the corresponding features are.

Why Absolute Difference?

- Simple and effective similarity measure
- Preserves information about which features are different

- Works well with the subsequent Dense layer for final classification

Training Strategy

Batch Processing: Processing images in batches (16 at a time) is more efficient than processing one by one, and provides better gradient estimates than using the entire dataset at once.

Epochs and Steps:

- Each epoch exposes the model to a fixed amount of data
- Multiple epochs allow the model to see data multiple times
- Steps per epoch control how much data the model sees in each epoch

Project Strengths and Limitations

Strengths:

1. **Balanced Data**: 50-50 split ensures unbiased learning
2. **Proper Preprocessing**: Normalization and resizing are correctly implemented
3. **Appropriate Architecture**: CNN is well-suited for image analysis
4. **Siamese Approach**: Good for signature verification tasks

Limitations and Improvements:

1. **Same Data for Validation**: Should use separate validation set
2. **Small Training Size**: Could benefit from more epochs and steps
3. **Simple Architecture**: Could add dropout, batch normalization, or more layers
4. **No Data Augmentation**: Could improve generalization with image augmentation

Conclusion

This project demonstrates a solid implementation of a Siamese CNN for signature verification. The code follows best practices for image preprocessing, network architecture, and training setup. The Siamese approach is particularly well-suited for this task because it learns to compare signatures rather than memorizing specific signature patterns, making it more robust to new, unseen writers.